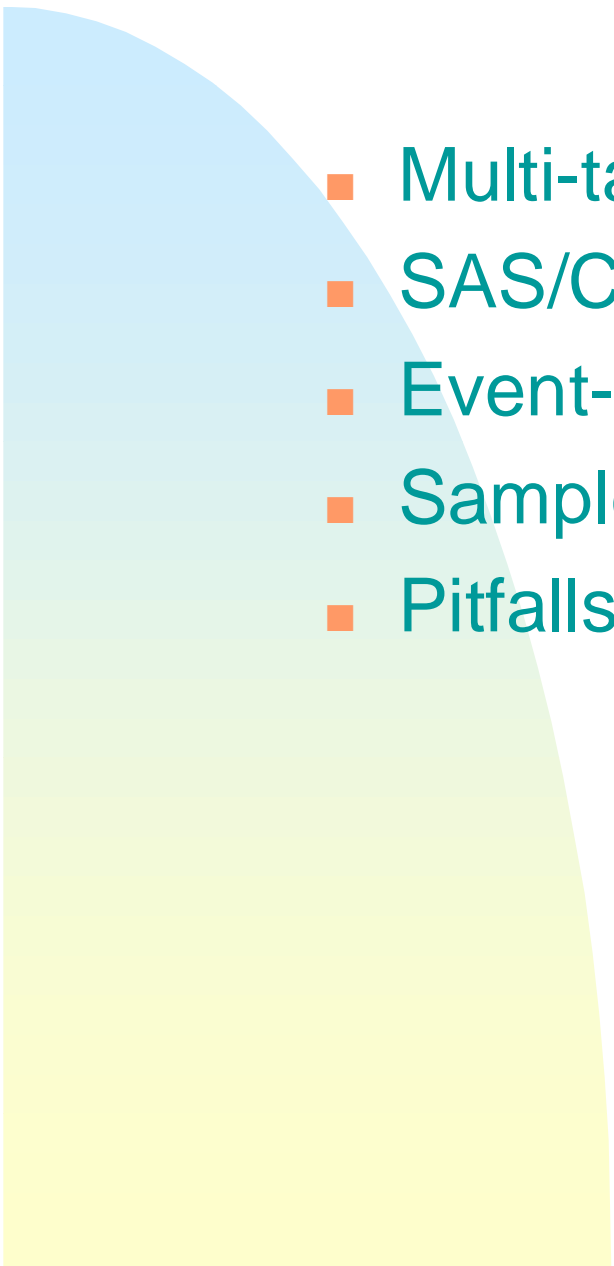


Traditional Multitasking Using Mainframe 'C'

Don Poitras
SAS/C Development
SAS Institute
sasdtp@sas.com

Share 1998 Winter Technical Conference
Session 5957

*Permission is granted to SHARE Inc. to copy, reproduce or republish this document in whole or in part for SHARE activities only.

- 
- Multi-tasking on MVS
 - SAS/C Multi-tasking features
 - Event-driven programming
 - Sample program
 - Pitfalls to avoid

Multi-tasking on MVS

- Multi-threading vs. Multi-tasking
- Assembler macros and control blocks
- Serialization and reentrant coding

Multi-threading vs. Multi-tasking

Multi-threading

- CICS and Windows 3.1
- All threads share heap, stack and PSW.
- “Cooperative” multitasking. One task can halt entire program.

Multi-tasking

- MVS, Win/NT and OS/2
- Each task has it's own PSW, stack and ability to get private heap.
- “Preemptive” multitasking. The OS gives each task a time slice. In multi-processing, each task could be running simultaneously.

Assembler macros and control blocks

■ Macros

- ◆ ATTACH/DETACH - start/stop a subtask
- ◆ WAIT/POST - wait for, or signal an event
- ◆ ENQ/DEQ - reserve or release a resource

■ Control blocks

- ◆ ASCB - Address Space Control Block
- ◆ TCB - Task Control Block
- ◆ RB - Request Blocks
 - ◆ PRB - Program Request Block
 - ◆ IRB - Interrupt Request Block
 - ◆ SVRB - SuperVisor Request Block for SVC routines

Serialization and reentrant coding

- Using the CS and CDS instructions.
 - ◆ Any global variable able to be updated by more than one subtask at the same time must be serially updated by some locking mechanism.
- The 'RENT' compiler option
 - ◆ All static and extern variables are `__rent` by default.
 - ◆ A PRV (psuedo register vector) is allocated for each load module. The same module attached twice would get two PRV's; each subtask see's only it's own static and extern variables.
- Marking a load module RENT. One copy loaded per address space. Better performance if in LPA.

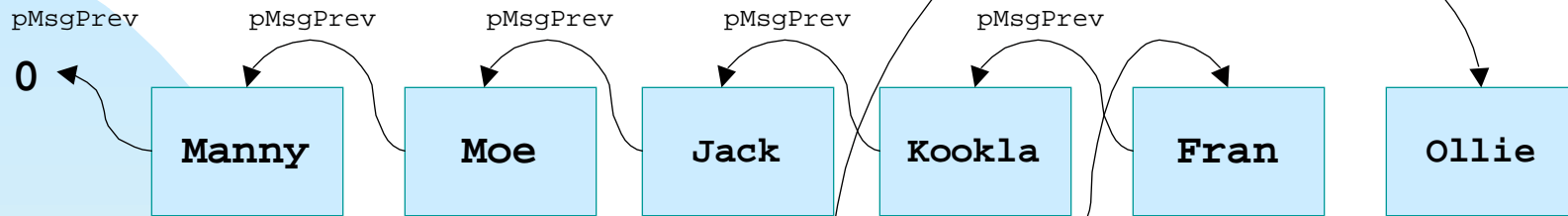
SAS/C Multi-tasking Features

- "Wrapper" functions
 - ◆ ATTACH/DETACH
 - ◆ ENQ/DEQ
 - ◆ POST/WAIT (WAIT1 or WAITM)
 - ◆ TPUT/TGET
- INDEP programs and the SPE Environment
 - ◆ STIMER/STIMERM

Event-driven programming

- Message passing
- Message queueing

A LIFO queue can be added to by multiple tasks and read by a single receiver by using the CS instruction.



pQueue->pMsgLast

/* put the message on the queue */

retry:

_ldregs(R1 | R3 | R15,

&pQueue->pMsgLast,

offsetof(MSGBLOCK, pMsgPrev),

pMsg);

L(2, 0, 0+b(1)); /* load old end pointer */

AR(3, 15); /* point to pMsgPrev of pMsg */

ST(2, 0, 0+b(3)); /* pMsg->pMsgPrev = pQueue->pMsgLast */

CS(2, 15, 0+b(1)); /* pQueue->pMsgLast = pMsg */

if (_cc() != 0)

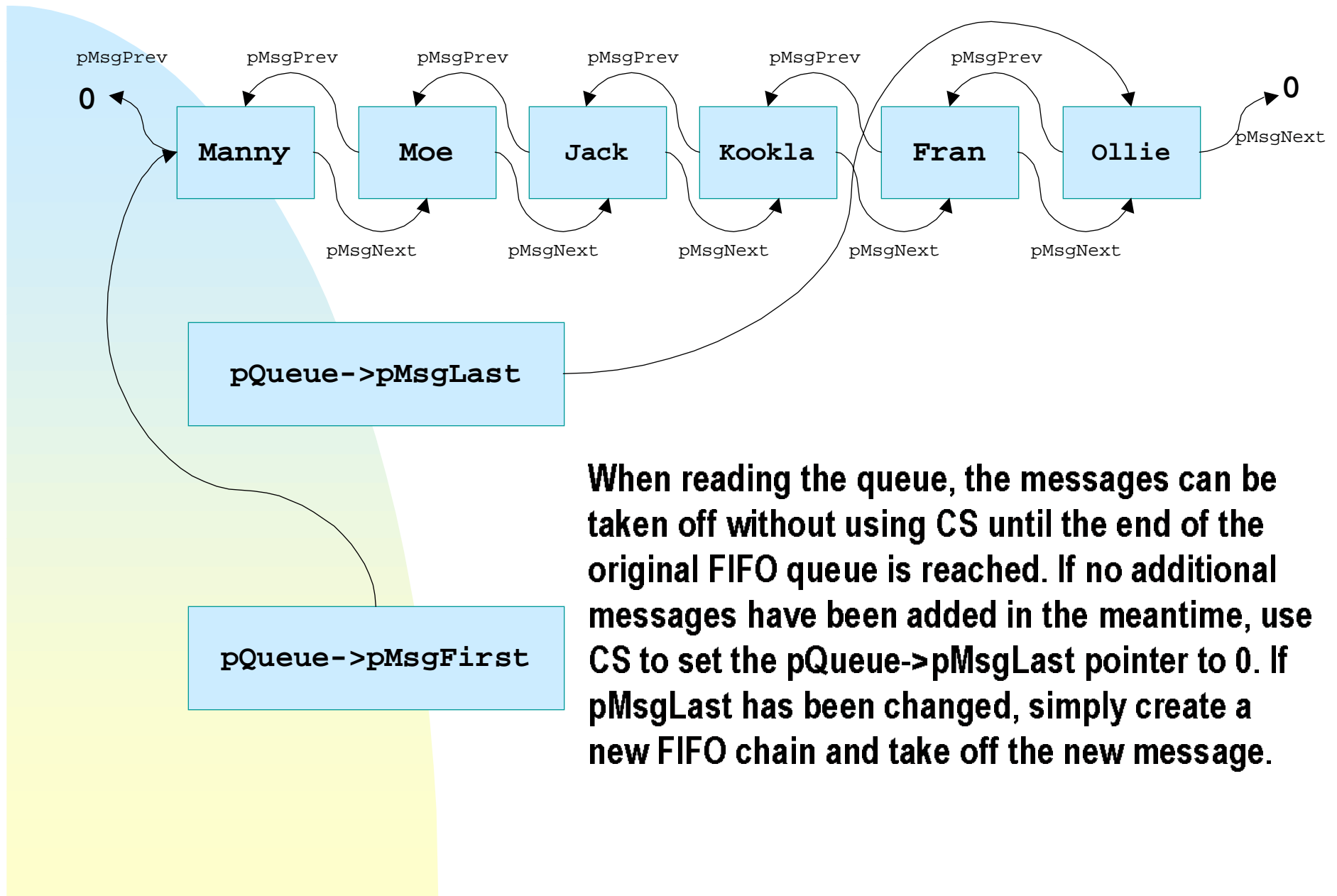
{

goto retry;

}

pMsg

However, a FIFO queue is more useful. Create a doubly-linked list by following the LIFO chain to the first message on the queue, filling in the pMsgNext as you go.



When reading the queue, the messages can be taken off without using CS until the end of the original FIFO queue is reached. If no additional messages have been added in the meantime, use CS to set the `pQueue->pMsgLast` pointer to 0. If `pMsgLast` has been changed, simply create a new FIFO chain and take off the new message.

Sample SAS/C multi-tasking program

This demo program illustrates some of the facilities available to application programmers to implement an event-driven multi-tasking application. Three generic sub-tasks and a special-use “timer” subtask are started and sent a few messages. After waiting for these messages to be processed, the main task then asks (politely) each of the subtasks to terminate. In the event that one or more subtasks does not respond within a reasonable period, they are shutdown forcibly.

In the following slides, we examine some details of the processing.

The "timer" subtask is started (attached.)

```
/*-----+
| attach the timer subtask                               |
+-----*/
iAttachRC = SubAttach(&pAnchor->stTimerTask, "SHRTIME");
if (iAttachRC != 0)
{
    printf("SubAttach of timer failed\n");
    exit(EXIT_FAILURE);
}

/*-----+
| SubAttach() is used to attach a sub-task.             |
+-----*/
static int SubAttach(PTASK pTask, PSZ pucModule) {
int rc;
    sprintf(pTask->stTaskArg.ucParm, "%08X =u", pTask);
    pTask->stTaskArg.sLen = 11;
    pTask->pstTaskArg = (void *) (0x80000000 |
                                (unsigned) &pTask->stTaskArg);
    rc = ATTACH(&pTask->pTaskTCB,
                _Aep, pucModule,
                _Aecb, &pTask->ulTaskEndECB,
                _Aparam, &pTask->pstTaskArg,
                _Aend);
    return (rc); }

```

All parms passed to a subtask must not be on the stack or extern/static variables.

Messages are sent to the “timer” task and the generic tasks.

```
/*-----+
| send a timer message to the first generic subtask |
+-----*/
stMsg.uiMsg = TIMERREQ;
time(&stTimerRequest.dWakeUpTime);
stTimerRequest.dWakeUpTime += 5; /* send the timer
                                message in 5 seconds */
stTimerRequest.pTask = pAnchor->pSubTaskChain;
memcpy(stMsg.ucData, &stTimerRequest, sizeof(TREQUEST));
WriteQueue(&pAnchor->stTimerTask.stQueue,
           4+sizeof(TREQUEST), (PUCHAR) &stMsg);

/*-----+
| send a few messages to each generic subtask |
+-----*/
pTask = pAnchor->pSubTaskChain;
for (i=0; i < NUM_TASKS; i++)
{
  int j;
  for (j=0x100; j < 0x103; j++)
  {
    stMsg.uiMsg = j;
    WriteQueue(&pTask->stQueue,
              4, (PUCHAR) &stMsg);
  }
  pTask = pTask->pNextTask;
}
```

Note that the message is copied, not passed to the recipient.

Limit the storage used by the subtask.

```
int _stack = 12288;
int _heap = 8192;
extern int _stkabv = 1;
extern int _stkrels = 1;
```

Limiting the size of global memory that all subtasks can access will reduce future headaches. Here, all global memory is accessed by the pAnchor pointer.

```
typedef struct _ANCHOR
{
    UCHAR          ucEye(|8|); /* eyecatcher - "ANCHOR " */
    PTASK          pSubTaskChain; /* points to first subtask cb on chain*/
    QUEUE         stMainQueue; /* message queue hdr for main task */
    TASK          stTimerTask; /* Timer subtask info */
    ULONG         ulSemLog; /* print log mutex semaphore */
    union
    {
        double dCDSAlign; /* used to force doubleword alignment */
        struct
        {
            PMSGBLOCK pMsgFirst; /* addr of first message in free pool*/
            ULONG ulRefNum; /* number of references to the pool */
        } stCDS;
    } uCDS;
    PMSGBLOCK pFreePool; /* points to the start of free pool*/
    ULONG ulFree; /* free messages in the pool */
} ANCHOR, *PANCHOR;
```

```
pTask = (PTASK) strtoul(argv[1], &pStopchar, 16);
pAnchor = pTask->pAnchor;
```

The subtask mainline is a “message loop”.

```
while (1)
{
    ReadQueue(&pTask->stQueue,
              &ulDataLength,
              (PUCHAR) &stMsg);

    switch (stMsg.uiMsg)
    {
        case SHUTDOWN:
            {   sprintf(buffer, "Subtask #%i, shutting down",
                        pTask->iTaskNum);
                Log(buffer);
                return (EXIT_SUCCESS); }

        case TIMERPOP:
            {   sprintf(buffer, "Subtask #%i, received timer pop",
                        pTask->iTaskNum);
                Log(buffer);
                break; }

        default:
            {   sprintf(buffer, "Subtask #%i, unknown message received %08x",
                        pTask->iTaskNum, stMsg.uiMsg);
                Log(buffer);
                break; }}}}
}
```

The “timer” subtask uses the SPE interface. A “minimal” C environment is created to reduce storage requirements. The timer program waits for requests from other tasks. If an OS timer (STIMER) has been started, the subtask must wait for either the STIMER to pop, or another timer request to be received.

```
/* set first ecb to wait on. This is the ReadQueue ecb */
aulecblList[0] = &pTask->stQueue.uCDS.stCDS.ulQECB;
/* set the stimer ecb in the wait list, and mark it as the end */
aulecblList[1] =
    (PECB) ((ULONG) &ulStimerECB | (ULONG) 0x80000000);

pStimer = bldexit(&StimerExit, ASYNCH);
ulStimerECB = 0; /* initialize the STIMER ecb */
pTimerChain = 0; /* initialize the timer chain */
while (1)
{ /* wait for a message or a timer pop */
    WAITM(1, aulecblList);
    ...
    if (!bLowestTimeSet) continue;
    ulTimeToWait = 100 * (dLowestTime - dCurTime);
    STIMER(REAL, pStimer, BINTVL, &ulTimeToWait);
}
static void StimerExit(void)
{
    POST(&ulStimerECB, 0);
}
```


Writing to a log file can cause problems if two subtasks attempt to write at the same time. A mutex semaphore can serialize the access to the file.

```
/* get the "Print Log Lock" before opening the file */
Lock(&stLogLockElement, &pAnchor->ulSemLog);
pstLogDCB = osdcb("log",
                 "lrecl=133,dsorg=ps,recfm=va,bufno=1,ncp=1",
                 0, 0);

osopen(pstLogDCB, "output", 0);
osput(pstLogDCB, abLine, strlen(abLine));
osclose(pstLogDCB, "disp");
/* release the lock. If other subtasks are waiting, the last
   one in will be posted to continue */
Unlock(&pAnchor->ulSemLog);
```

This code is taken from the ESA/390 Principles of Operation

```
void Lock(PLOCK pLock, PULONG pQueue)
{
    pLock->ulLockECB = 0; /* Clear our lock element ECB */
    _ldregs(R1 | R2,      /* Load element and header address */
           pLock, pQueue);
    LNR(0, 1);           /* Force R0 negative as a flag */
    XR(3, 3);           /* Clear R3 for use in the CS instruction*/
    BASR(14, 0);        /* set address for retry */
    CS(3, 0, 0+b(2));   /* Set the header to a negative value
                        if the current header is 0 */

    if (_cc() == 0)
    {
        return;         /* If successful, exit */
    }
    _ldregs(0);         /* Tell the compiler we're starting
                        an assembler sequence */
    ST(3, 0, 4+b(1));   /* Save the address of the prior locker
                        in my lock element */
    CS(3, 1, 0+b(2));   /* Store our element address into the
                        header (this time it's not negative)*/
    LA(3, 0, 0+b(0));   /* Clear R3 in case we need to try the
                        first CS again. */
    BCR(7, 14);         /* If the store was interrupted, go back
                        to the first CS */

    WAIT1(&pLock->ulLockECB);
}
```


At shutdown, care needs to be taken to avoid getting stuck waiting for all subtasks to complete.

```
/*-----+
| Wait for all subtasks to end. Detach the TCBs when the |
| shutdown ecbs are posted.                               |
+-----*/
while (1)
{
int bWaitSomeMore;

    signal(SIGALRM, &TimeAlarm);/* Catch SIGALRM signal.      */
    alarm(10);                  /* Wait max. ten seconds for some
                                task(s) to end */
    sigemptyset(&stNoSigs);     /* Set no sigs blocked for suspend. */
    /* wait for one or more shutdown ecbs to be posted, or for the
       alarm signal */
    ecbsuspend(&stNoSigs, 1+NUM_TASKS, pShutdownECBs);
    alarm(0);                   /* Cancel alarm. */
    signal(SIGALRM, SIG_DFL);   /* Restore default alarm handling. */
}
```

Some pitfalls to avoid

- Shared resource conflicts
 - ◆ Use ENQ/DEQ or WAIT/POST locks.
 - ◆ Beware the deadly embrace.
- Common abends
 - ◆ 0C4 - Subtask passed bad pointer.
 - ◆ 23E - Bad TCB pointer in DETACH.
 - ◆ 13E/33E - Subtask detached before it returned.
 - ◆ 43E - ECB specified at ATTACH was invalid when the task ended.
 - ◆ C03 - Subtask ended without closing DCB.

Further information

- Information about SAS/C
 - ◆ http://www.sas.com/software/sas_c/
 - ◆ http://www.sas.com/software/sas_c/whitepapers/doc/lrv2ch4.html#lr2multi
- IBM Manuals
 - ◆ <http://ppdbooks.pok.ibm.com:80/cgi-bin/bookmgr/bookmgr.cmd/books/IEA1A702>
 - ◆ <http://ppdbooks.pok.ibm.com:80/cgi-bin/bookmgr/bookmgr.cmd/books/IEA1A802>
- Questions or comments for me
 - ◆ <mailto:sasntp@sas.com>